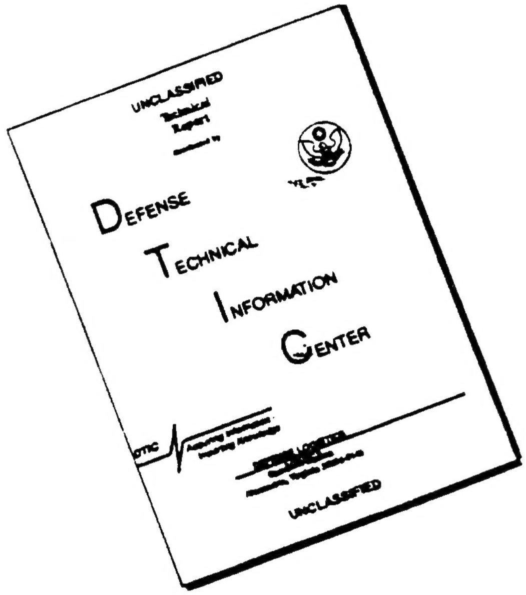
ARMY RESEARCH LABORATORY

An Efficient Method for Parsing Large Finite Element Data Files

Dale Shires

ARL-TR-974 March 1996

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

NOTICES

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1996	Final, June 1995–September 1995

4. TITLE AND SUBTITLE

An Efficient Method for Parsing Large Finite Element Data Files

5. FUNDING NUMBERS

PR: 1L162618AH80

6. AUTHOR(S)

Dale Shires

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

U.S. Army Research Laboratory
ATTN: AMSRL-SC-SM
Aberdeen Proving Ground, MD 21005-5067

8. PERFORMING ORGANIZATION REPORT NUMBER

ARL-TR-974

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The FORTRAN language has been and continues to be one of the most heavily used languages for implementing complex numerical algorithms. It is extremely efficient, and the straightforward constructs of the language allow today's FORTRAN compilers to take full advantage of the most novel parallel and vector computers. Many of these algorithms require large data sets to be read. This is the one task in which FORTRAN is somewhat lacking. While the main computation task should be left in FORTRAN, parsing tasks should be performed by applications with greater character stream access. Accessing and processing this character stream that is formed while the file is being read is known as parsing. This paper describes the tools available on most UNIX systems which can produce very fast parsers, the underlying methods employed by these tools, and ways these tools can be integrated with FORTRAN numerical solvers.

14. SUBJECT TERMS

parsing; finite element method

15. NUMBER OF PAGES

27

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

INTENTIONALLY LEFT BLANK

Contents

INTENTIONALLY LEFT BLANK

List of Figures

List of Tables

INTENTIONALLY LEFT BLANK

1 Introduction

FORTRAN remains the language of choice for many complex numerical algorithms. The motivations behind the development of the language help to explain its longevity. Researchers early in the computer revolution were confined to writing numerical codes in assembly language. This practice required detailed knowledge of the algorithm as well as assembler and computer architecture specifics such as number of registers, memory structure, etc. The development of the FORTRAN language provided a watermark for both programming language and compiler designers. Advances in compiler design provided compiler writers the first opportunity to take a program written in a high-level language and generate assembly code of a caliber often exceeding hand-coded assembly. Codes began to be written in FORTRAN, at which point the computer specifics could be left to the compiler writers.

FORTRAN remains popular today because it is highly efficient. The time required to execute many of these numerical codes is most often dominated by one or two small loops which perform the vast majority of the overall work of the algorithm. It is not uncommon to find one or two loops in these codes which consume upwards of seventy percent of the overall execution time. FORTRAN is very efficient at processing these loops. The simplicity of the language's loop structure is one of the main factors allowing for highly-optimized compiler-generated code. These loops can be executed with great speed with little overhead being incurred due to language constructs.

While it is very efficient at number crunching, FORTRAN is somewhat lacking when it comes to file input and output. Often associated with these numerical codes are very large input files or data decks. The problem of interest for this research team provides an excellent example. This team is particularly involved with manufacturing simulation dealing especially with composite materials. To this end, two algorithms have been developed. One is a new variant of the control volume finite element algorithm to simulate the isothermal flow of resin in the resin transfer molding (RTM) composite manufacturing process [1]. The other is an implicit time-dependent pure finite element methodology for RTM flow simulation [2]. The majority of the work in both algorithms is performed in a few small FORTRAN loops. These codes perform very well on the new pipelined architecture found in the Silicon Graphics Power Challenge computer. However, parsing the input files is annoyingly slow and at times convoluted.

This speed can be increased by taking input and output tasks away from languages like FORTRAN, which are limited in this area, and moving them to more robust byte-stream languages and libraries like those found and written in C. Furthermore, formalizing on one simple yet robust input format will also allow for faster reading. Combining regular expressions and a context-free grammar describing the structure of the input file makes it possible to create a deterministic finite automata for pattern recognition and a parser to interpret the structure of the file. Parsing of the input file is then bounded by $O(n)$, where n is the size of the input deck. The techniques

mentioned previously were implemented to reduce the time required to parse finite element input files. This paper describes the implementation steps and the overall results of using this parsing technique.

2 Elements of fast parsing

There are several key issues which must be addressed in the course of defining a parsing strategy. What are the basic items in the data file? What is the basic structure of the data file? These issues are not unlike those historically encountered in the development of parsing strategies for compilers. They involve:

- *Defining the basic units of the data file.* In this case, these items include instances of real numbers, integers, and character strings.
- *Formalizing a description of the format of the data.* This is done by defining a grammar for the input data.
- *Establishing what to do with the data as they are being read.* This requires establishing data structures and actions.

Often the best way to overcome a multifaceted problem such as this is to use the divide and conquer approach. This approach calls for us to solve each of these parts of the main problem separately. The methods are described in the following sections.

2.1 Lexical analysis

Lexical analysis is the process of identifying the basic units of the data file. This process is accomplished by scanning the input stream, recognizing patterns in the data, and converting these patterns into tokens. These tokens are basically some classification for the patterns. For example, the sequence of characters "program" forms a **string** token and the sequence of numbers 531 forms a **number** token. These classifications are arbitrary and must be defined by the user.

The process of building a pattern recognizer requires the construction of a transition diagram referred to as a finite automaton. These finite automatons are state-transition diagrams. They tell the controlling algorithm how to act based on the current state it is in and on the next character in the input stream. The finite automaton in figure 1 can accept a string with zero or more x characters ending with the sequence yz.

A finite automaton can be either deterministic or nondeterministic. Nondeterministic automatons allow more than one transition out of a state on the same input symbol whereas deterministic automatons do not. There is a space-time tradeoff between the two approaches. In general, deterministic finite automata allow for faster recognizers but require more space to define.

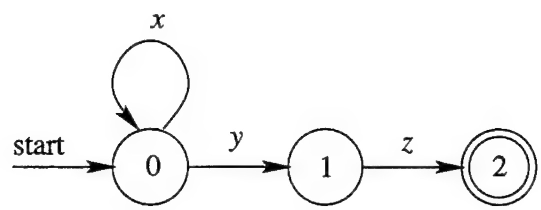


Figure 1: Finite automaton recognizing a string with zero or more x characters ending with the character sequence yz.

Several tools have been created to make the task of constructing a lexical analyzer easier. One such tool widely available on UNIX machines is Lex. Lex accepts pattern definitions and generates a deterministic finite automaton for the input stream. Lex users may also supply code fragments to complete various operations once a token is recognized. For example, a code fragment may be written which converts all strings to their upper-case equivalents.

Specification of patterns is accomplished through the use of regular expressions. UNIX regular expressions have existed for some time and are used in a variety of operating system utilities such as *awk*, *vi*, and *sed*. Regular expressions provide a robust method for specifying patterns in data. Indeed, these regular expressions are mathematical objects, and as such, may consist of the empty set, a single character, unions and concatenations of regular expressions, or repetitions of regular expressions [3]. Some of the basic symbols used in defining regular expressions in Lex are listed in table 1. * For example, the regular expression `[+-]?[0-9]+` can match `-241`, `+023`, `51`, etc. Finite element input files contain several entities that must be

Table 1: Some regular expression operators of Lex.

Symbol	Meaning
.	Matches any character up to, but not including, a new line.
[]	Matches any of the characters listed.
?	The previous regular expression is optional.
$*$	Zero or more repetitions.
+	One or more repetitions.
|	One or the other.
()	Grouping of expressions.

recognizable. They contain character strings which signify which part of the file is currently being read (nodes, elements, etc). They contain both integer and real numbers. They also contain delimiter characters, such as tabs or spaces, which segregate the items. They also often contain comments. All of these, with the exception of real

*A complete listing of regular expression operators may be found in [3].

numbers, are fairly easy to define with regular expressions. FORTRAN real numbers are somewhat more involved and require several regular expressions to describe all the possible formats they may take. Table 2 lists the regular expressions used to define some of the items encountered in finite element mesh files.

Table 2: Regular expressions for items in finite element mesh files.

Token	Regular expression	Example
comment	`"*".*`	`* File:mesh.data`
string	`[a-zA-Z]+`	`nodes`
integer	`[-+]?[0-9]+`	`+641`
real	`[-+]?"."[0-9]+`	`.341`
	`[-+]?[0-9]+"."`	`-423.`
	`[-+]?([0-9])+"."[0-9]+`	`+35.501`
	`[-+]?([0-9])*"."([0-9])*[eE][-+]?([0-9])+`	`9.34e-5`
	`[-+]?([0-9])*[eE][-+]?([0-9])+`	`34e8`

2.1.1 Lexical analysis with Lex

The Lex specification file is given in appendix A. The beginning of the file lists several libraries that need to be included for various purposes such as string manipulation and input/output operations. Also listed are definitions for various local and global variables and function prototypes. Following this is the list of regular expressions for the finite element data file. This section follows the %} and ends with the first %%.

White space is defined as any space, tab, or newline character. The definitions for letters and digits are straightforward. Integers have an optional sign followed by one or more digits. There are various definitions for real numbers to correspond with all allowed FORTRAN real formats. Strings are defined to be sequences of letters. Finally, a comment is defined to start with an $*$ and comprise all characters until the end of the line.

Next comes a list of actions that are to be performed when the regular expressions are matched. For integers, the string of characters is converted to an integer whose value is stored for the parser to use. The token **integer** is returned to the parser. For real numbers, a similar action is taken with a **real** token being returned to the parser. White space and comments result in no actions. All strings are first converted to upper case. A function is then called which scans a list of keywords, and if the string is a reserved word or keyword, returns a token for the keyword. Finally, any unmatched characters result in an error message being displayed.

Following the second %% and continuing until the end of the file are the supporting functions. These functions perform various tasks such as converting strings from lower to upper case and checking a string to see if it is a keyword.

2.2 Syntax analysis and parsing

The input deck for the executing code must adhere to some rigid format to facilitate quick scanning. This format, or syntax, is best defined through the use of a context-free grammar, or grammar for short. A grammar naturally describes the syntactical structure of a language. Grammars can be very complex because of this. Indeed, they are most often used to define elaborate hierarchical and recursive constructs in programming languages. In this case, the format for an input deck, as well as the defining grammar, can be very simple. Context-free grammars consist of four components:

1. A set of tokens, or terminal symbols. These are the items recognized and returned by the lexical analyzer.
2. A set of nonterminals.
3. A set of productions. These productions consist of a nonterminal on the left side, an arrow, and then a sequence of nonterminals or tokens on the right side.
4. A nonterminal designated as the start symbol.

Historically grammars are specified by listing their productions with the start symbol listed first. Productions define the valid orderings of tokens in the file. Digits and boldface strings such as **nodes** are considered to be terminals. Italicized names are nonterminals and any nonitalicized names or symbols are tokens. If the nonterminal on the left has more than one production, the right sides may be grouped and separated with the | symbol.

For example, the grammar below may derive one item of the set of domestic animals {dog, cat} or one item of the set of wild animals {racoon, wolverine, bear}.

$$\begin{aligned} \textit{animals} &\rightarrow \textit{domestic} \mid \textit{wild} \\ \textit{domestic} &\rightarrow \textbf{dog} \mid \textbf{cat} \\ \textit{wild} &\rightarrow \textbf{racoon} \mid \textbf{wolverine} \mid \textbf{bear} \end{aligned}$$

The structure of the finite element input deck can be of a simplistic nature. For the isothermal filling algorithm, the vast majority of the file will be entries defining the grid points of the mesh and corresponding connectivity of these points. These entries are often referred to as nodes and elements, respectively. Other entries, such as material descriptors, may also be required. General purpose structural analysis programs have more functionality and usually support many data descriptors. For example, NASTRAN * supports over 100 data card descriptors [4]. Since we are more concerned with flow simulations, we focus on the two descriptors comprising the bulk of our data files. However, parser construction through grammar specification is the same for both large and small input formats.

*NASTRAN is a registered trademark of the National Aeronautics and Space Administration.

A grid point, or node, in a finite element mesh is defined by three points: the x, y, and z locations in 3-D space. Another identifier, the node number, is also required to later define connectivity. To specify a node we therefore need a sequence of four tokens:

integer real real real

to denote the node's identification number and its x, y, and z values, respectively. Elements, which in this case are triangular with material and thickness data, can be described with a sequence of six tokens:

integer integer real integer integer integer

to denote the element number, material identifier, thickness of the element, and nodes which comprise the element, respectively. Defining two more terminal symbols will also be necessary for our grammar. They will make it more readable and solve the problems which would arise by adding more data cards with similar defining sequences of numbers. Therefore, we also define the tokens **nodes** and **elements**. The definition of elements refers only to triangular elements. Higher order elements and mixed element types which may require more data variables can easily be incorporated by creating new tokens followed by the required sequences of integers and reals.

Grammars, in their own right, do not actually parse a file. Grammars are used to define the way in which the parsing machine is constructed. Parsing is actually done in a linear fashion by constantly processing tokens from the lexical analyzer and determining if the token stream can be derived from the grammar. There are several parsing strategies, with each one having several advantages and disadvantages. Often which strategy to implement depends on characteristics of the defining grammar. There are several areas of concern which arise depending on which type of parser is being used. Some parsers do not accept left recursive grammars. A left recursive grammar, such as $A \to A + B$ would not be acceptable to a top-down parser because it would lead to an infinite loop as A continually derives A. Some parsers cannot work with ambiguous grammars. The grammar

$$E \to E + E \mid E * E \mid \mathbf{a}$$

is ambiguous because it can derive the string $\mathbf{a} + \mathbf{a} * \mathbf{a}$ in two different ways:

$$\begin{array}{ll}
E \to E + E & \qquad E \to E * E \\
\quad \to \mathbf{a} + E & \qquad \quad \to E + E * E \\
\quad \to \mathbf{a} + E * E & \qquad \quad \to \mathbf{a} + E * E \\
\quad \to \mathbf{a} + \mathbf{a} * E & \qquad \quad \to \mathbf{a} + \mathbf{a} * E \\
\quad \to \mathbf{a} + \mathbf{a} * \mathbf{a} & \qquad \quad \to \mathbf{a} + \mathbf{a} * \mathbf{a}
\end{array}$$

Irregardless, the process of building a parser can be a laborious one requiring the compiler writer to compute many complicated sets and tables. A complete discussion of parsing and syntax analysis is beyond the scope of this paper and left to the reader.* Special computer programs, called parser generators, have been written to mitigate some of this complexity. They take grammars as input and construct the set of parsing action tables. These utilities are very helpful in instances where the defining grammar may change or be augmented, as is true in this case. The most widely available parser generator is Yacc (yet another compiler-compiler), and it is used to generate the parser for this grammar.

The parser generated with Yacc is termed an LALR parser. The "LA" stands for lookahead and the "LR" for left-to-right scanning of the input, rightmost derivation in reverse. This parser has four actions it can perform: accept the input, indicate an error, shift, or reduce. The input is accepted if it can be derived from the grammar, otherwise an error is reported. Shifts are the most common operation and are performed while the input is being parsed. A reduction is performed when the right hand side of a grammar production is recognized. Consider a simple grammar with one production $S \rightarrow \mathbf{a\ b\ c}$ and an input stream *abc*. The parsing table for this grammar with states and actions is shown in table 3. The actions taken by the parser are shown in table 4. The $ represents the end of input.

Table 3: Parsing table.

state	action				goto
	a	**b**	**c**	$	S
0	s2				1
1				accept	
2		s3			
2			s4		
4				reduce	

Table 4: Parser actions.

stack	input	parser action
0	*abc*$	shift 2
$0a2$	*bc*$	shift 3
$0a2b3$	*c*$	shift 4
$0a2b3c4$	$	reduce $S \rightarrow \mathbf{a\ b\ c}$
$0S1$	$	accept

We start in state 0 with an a as the next symbol in the input stream. According to the table, state 2 is shifted onto a run-time stack. In state 2 with b the next symbol, the action is to shift state 3. State 4 is then shifted onto the stack and all data have been shifted. In state 4 with the end of file marker we reduce by the rule $S \rightarrow \mathbf{a\ b\ c}$. This pops three states off the stack, leaving us in state 0 with the symbol S on the stack. We then go to state 1 with the end of file marker still the next symbol in the input stream. At this point, the parser accepts the input.

LALR parsers can accept ambiguous grammars. Yacc provides mechanisms such as precedence operators to preclude ambiguity. During its final stage of processing, Yacc will actually report the number of ambiguities it encountered and could not

*For additional information regarding issues in parsing and syntax analysis, see [5].

resolve. These errors are either shift-reduce errors or reduce-reduce errors. A shift-reduce error occurs when the parser has reached a state where it could either shift the next input symbol or reduce a right hand side. Reduce-reduce errors occur when the parser reaches a state where two possible reductions could be performed.

The grammar for the finite element input files need not be as involved as those for some programming languages. Accordingly, rather than trying to use disambiguating rules, the grammar should be designed so that there are no ambiguous rules. It makes sense to group similar data items in the file. The best way to do this is to partition the data file into logical blocks of similar items. The data is grouped by using the **nodes** and **elements** tokens. These tokens inform the parser of what to expect next in the file and allow the data to be grouped in a manner such as:

```
NODES
 ⋮
ELEMENTS
 ⋮
```

Given all of the above information, figure 2 lists a first try at specifying a grammar for the nodes and elements of the finite element input file.

startpoint	→	*items*
	|	*startpoint items*
items	→	**nodes** *node_list*
	|	**elements** *element_list*
node_list	→	**integer real real real**
	|	**integer real real real** *node_list*
element_list	→	**integer integer real integer integer integer**
	|	**integer integer real integer integer integer** *element_list*

Figure 2: A possible grammar specification.

2.2.1 Structuring the grammar for Yacc

The grammar given in figure 2 is easy to understand. The start symbol is called *startpoint*. This nonterminal can derive one item, or many items by recursively calling itself. This is a left-recursive rule. Notice also that right-recursive rules are used in figure 2 to specify the list of nodes and list of elements. LALR parsers can accept grammars which have both left and right recursive rules. These rule structures are often used for specifying lists. The list of items includes nodes and elements. The list of nodes and elements specify the sequence of tokens that should be encountered. The lists of nodes and elements continue as long as a valid sequence of **real** and **integer** tokens are read.

While both left and right recursion may be employed, there is a significant reason for choosing left recursion. The reason mainly involves how Yacc builds the parsing engine. At first sight, the right-recursive rule would seem to be more intuitive. The input file is read top-down, left to right. Once the **nodes** token is read, the **integer** token should follow as well as three **real** tokens. The process then resumes with a new list of nodes.

However, since this rule is right recursive, the stack maintained by Yacc will continually grow until the **elements** token is reached. It is only at this point that the rule will be reduced and items will be popped from the stack. Large files will result in a stack that grows very quickly. For example, a file approximately 3.7 Mbytes in size was parsed using Yacc and the grammar in figure 2. As reported from the Silicon Graphics IRIX operating system utility *osview*, this parsing process required 31 Mbytes to be allocated from free memory space.

In contrast, left recursive rules limit stack size by reducing right hand sides more quickly. The states are popped from the stack during these reductions and the stack is kept to a small size. With this in mind, the grammar of figure 2 was reconstructed and is shown in figure 3. This process only required 1 Mbyte to parse with the final outcome the same as the right-recursive parse.

startpoint	→	*items*
	|	*startpoint items*
items	→	**nodes** *node_list*
	|	**elements** *element_list*
node_list	→	**integer real real real**
	|	*node_list* **integer real real real**
element_list	→	**integer integer real integer integer integer**
	|	*element_list* **integer integer real integer integer integer**

Figure 3: Restructured grammar using left-recursive rules.

The left-recursive rules allow the first production of the *node_list* nonterminal to be reduced for the first node encountered in the file. All subsequent nodes in the file are then reduced by the second *node_list* right hand side. In this fashion, there will never be more than four items shifted onto the stack between reductions. In constrast with the first parser, the parser generated from the left-recursive grammar consumes very little memory. The state transitions used by the Yacc parser engine are available for analysis. Using the command *yacc -d filename* produces a file named "y.output" containing the transition rules. Careful study of "y.output" files produced with the right and left-recursive rules will clearly demonstrate the differences in the parser engines.

2.2.2 Parsing with Yacc

The Yacc specification file is given in appendix B. The beginning of the file is similar to the Lex specification where included library routines are listed. Following this a list which defines the tokens. Some tokens have attributes associated with them. For instance, the token **integer** should have some integer value associated with it. This association of tokens with actual data is accomplished using the C structure feature. The lexical analyzer will set the integer attribute in a code fragment upon encountering an integer, the real attribute upon encountering a real, etc. This structure is created with the `%union` statement. The variable `yylval` assumes this structure. Accordingly, upon encountering an integer in the input data, the lexical analyzer can set `yylval.integer` equal to the actual encountered integer. The start point is defined to be *startpoint*.

Enclosed by the `%%` symbols is the context-free grammar in Yacc syntax. The grammar is identical to that given in figure 3 with a minor difference. Actions are placed inside {} symbols. As an example, consider the *node_list* productions. The actions involve actually storing the data encountered during the parse into some structure for later use. In this case, the numbers being read are stored into arrays. The `$` allows access to the values that were assigned in the lexical analysis section. In the statement

```
node_list :  INTEGER REAL REAL REAL
```

the actual integer value associated with the **integer** token may be accessed by using the `$1` operator since it is the first token to the right of the colon. The real values associated with the **real** tokens are accessible by using `$2`, `$3`, and `$4`. The correction by -1 for the arrays is attributable to the difference in the way C and FORTRAN handle array storage. Following the second `%%` to the end of the file are various supporting functions.

3 Combining the parts

The Lex and Yacc specifications have been described in some detail. The only remaining point of discussion is how to properly tie these items together. Since most of the computing is done in FORTRAN, the driver for the parser is also given in FORTRAN. The code for this routine is listed in appendix C. C and FORTRAN code can easily be combined. The main concern is making sure that the variable types match between the two languages. Appendix D lists the header file for the Lex and Yacc routines which defines the C structure to match variables in the FORTRAN structure.

To compile the Yacc specification file, issue the command *yacc -d filename*. This creates a file named *y.tab.c*. The *-d* option instructs Yacc to generate a file named *y.tab.h* containing token definitions which must be included into the Lex specification

file. To compile the Lex specification file, issue the command *lex filename*. This produces a C file named *lex.yy.c*. The files *lex.yy.c* and *y.tab.c* must be compiled separately from the FORTRAN files by issuing the command *cc -c lex.yy.c y.tab.c*. This will generate files which can be linked and loaded by the FORTRAN compiler. The final command to create the executable is then *f77 y.tab.o lex.yy.o driver.f*.

4 Results

This section lists some parsing time results for the LALR parser generated with Lex and Yacc and also some comparisons to a parsing system written in FORTRAN. The FORTRAN parsing technique required one line to be read at a time from the input file. This line was then searched by a routine which identified tokens in the line.

Two files were used as test cases. The first was a mesh of a bridge truss having 5,325 nodes, 10,898 triangular elements, and 865,511 total characters. The second file was the mesh of a component of the RAH-64 Comanche helicopter. This mesh comprised 23,348 nodes, 45,990 triangular elements, and 3,697,579 total characters.

Parse times were averaged over three trials. The trials were performed on a Silicon Graphics Computer Systems Power Challenge 75-MHz R8000 processor. Table 5 lists the results.

Table 5: Results of parsing trials.

File (size)	Parse time (in seconds)	
	FORTRAN	LALR (Lex & Yacc)
Bridge truss (845 Kbytes)	43.98	2.83
RAH-64 Comanche (3.69 Mbytes)	185.78	11.79

Table 5 shows some rather dramatic results. The parser generated by Lex and Yacc was able to parse the input files approximately 15 times faster than the corresponding FORTRAN parser. The multiple scanning used by the FORTRAN parsing method severely degrades that parser's performance.

5 Conclusion

Lex and Yacc provide effective tools for implementing LALR parsers quickly and easily. These tools promote parser expandability and impart a logical nature on the entire parser construction process. Most importantly, the LALR parsers generated by Lex and Yacc are extremely efficient. This efficiency easily supersedes that of many other more contrived methods. This reduced parse time is notable and worth pursuing in virtually all data file parsing tasks.

References

[1] R. Maier. A fast numerical method for isothermal resin transfer mold filling. Technical Report 94-028, U. S. Army High Performance Computing Research Center, Minneapolis, 1994.

[2] R. Mohan, N. Ngo, K. Tamma, and K. Fickie. On a pure finite element based methodology for resin transfer mold filling simulations. In R. Lewis and P. Durbetaki, editors, *Numerical Methods for Thermal Problems*, volume IX, pages 1287–1310, Swansea, UK, July 1995. Pineridge Press.

[3] T. Mason and D. Brown. *Lex & Yacc.* O'Reilly & Associates, Inc., Sebastopol, California, 1990.

[4] The MacNeal-Schwendler Corporation, Los Angeles. *MSC/NASTRAN Handbook for Linear Analysis*, August 1985.

[5] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, March 1988.

A Lex specification

```
%{
/*

File: lex.spec
Description:
   Specify the lexical analysis component for parsing.

   Fortran real numbers may be represented in one of the
   three following formats:

      sww.ff   Basic real constant.
      sww.ffEsee  Basic real constant followed by real exponent.
      swwEsee   Integer constant followed by real exponent.

   The following real number regular expressions apply:

      realconst1 -> -.5 .62 +.123
      realconst2 -> -5. +69. 0.
      realconst3 -> -6.2 +9.3 82.3
      realconst -> Any of the above constructs.
      realconstwexp -> -.3E1 +9.12E-4
      intconstwexp -> -3E-5 9E4
External global variables:
   meshdata_ - Structure to hold data read.  Used for summary printout.
   startTime - Time parsing began.  Used for summary printout.
   numberYaccErrors - Number of parsing errors discovered in Yacc.
Local global variables:
   keywordName - Holds the symbolic name of keyword found.
   numberLexErrors - Counter for number of lexical errors.
Functions (functions appear in alphabetical order):
   ConvertToUpperCase - Convert a string to upper case.
   IsKeyword - True if a lexeme is a keyword, false otherwise.
   ReportError - Show error message when a lexical error is discovered.
   yywrap - Report parsing statistics and perform last steps before the end
      of parsing.

*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <limits.h>
#include <sys/types.h>
#include <time.h>
#include "parser.h"
#include "y.tab.h"

/* External variables: */

extern MeshStruct meshdata_;
extern time_t startTime;
extern int numberYaccErrors;

/* Local variables: */

static int keywordName;
static int numberLexErrors = 0;

typedef enum {false, true, FALSE=0, TRUE} boolean;

/* Function prototypes: */
```

```
static void ConvertToUpperCase();
static boolean IsKeyword();
static void ReportError();
int yywrap();

%}

ws              [ \t\n]
letter          [a-zA-Z]
digit           [0-9]
integer         [-+]?{digit}+
realconst1      [-+]?"."{digit}+
realconst2      [-+]?{digit}+"."
realconst3      [-+]?({digit})+"."{digit}+
realconst       {realconst1}|{realconst2}|{realconst3}
realconstwexp   [-+]?({digit})*"."({digit})*[eE][-+]?({digit})+
intconstwexp    [-+]?({digit})*[eE][-+]?({digit})+
real            {realconst}|{realconstwexp}|{intconstwexp}
string          {letter}+
comment         "*".*

%%

{integer}       { yylval.integer = atoi(yytext); return INTEGER; }
{real}          { sscanf(yytext, "%lf", &yylval.real); return REAL; }
{ws}            { /* Consume white space without action. */ }
{string}        { ConvertToUpperCase();
                  if (IsKeyword())
                     return keywordName;
                  else {
                     ReportError();
                     return REAL;
                     }
                }

{comment}       { /* Take no action for comments. */ }
.               { ReportError();
                  /* Return an arbitrary token to let the parser continue. */
                  return REAL;
                  }

%%

/* Define an array containing the list of keywords. */

static struct keywordsStruct {
   char *name;
   int symbolicName;
   } keywords[] = {
      "NODES", NODES,
      "ELEMENTS", ELEMENTS,
      (char *)NULL, 0
      };


/* ========================================================================
   ConvertToUpperCase
   Purpose:
      Convert all letters in yytext to lower case.
   Global variables:
      yytext - The lexeme matched from the regular expression.
         All characters in yytext are converted to upper case.
      yyleng - The length of the lexeme.
   Local variables:
```

```
     i - Counter.
---------------------------------------*/

static void ConvertToUpperCase()
   {
   int i;

   for (i=0; i<yyleng; i++)
      yytext[i] = toupper(yytext[i]);

   } /* end ConvertToUpperCase */


/* ==========================================================================
   IsKeyword
   Purpose:
      Return true if the lexeme is a keyword, false otherwise.
   Global variables:
      yytext - The lexeme matched from the regular expression.
      yyleng - The length of the lexeme.
   Local variables:
      ptr - Pointer to keyword list.
   Returned value:
      found - False if the lexeme is a keyword, true otherwise.
---------------------------------------*/

static boolean IsKeyword()
   {
   boolean found = false;
   struct keywordsStruct *ptr = keywords;

   ptr = keywords;
   while ((!found) && (ptr->name != NULL)) {
      if (strncmp(yytext, ptr->name, yyleng) == 0) {
         found = true;
         keywordName = ptr->symbolicName;
         }
      ++ptr;
      }

   return(found);

   } /* end IsKeyword */


/* ==========================================================================
   ReportError
   Purpose:
      Report when a lexical analysis error was discovered (no pattern matching
      rule was found) and increment the error counter.
   Global variables:
      numberLexErrors - Counter for the number of lexical errors discovered.
      yylineno - Parser current line number.
      yytext - The matched pattern.
---------------------------------------*/

static void ReportError()
   {

   ++numberLexErrors;
   printf("Invalid item found at or near line %d: %s\n", yylineno, yytext);

   } /* end ReportError */
```

```
/* ===========================================================================
   yywrap
   Purpose:
      Perform wrap up on end-of-file.  Currently this prints statistics
      about the number of nodes, elements, etc.
      Statistics are only printed if no errors were found.
   Local variables:
      endTime - Time at which this function is called.
   Global variables:
      numberLexErrors - Number errors encountered during lexical analysis.
      numberYaccErrors - Number errors encountered during parsing.
      startTime - Time at which parsing began.
      meshdata_ - The structure to store nodes, elements, etc.
   Returned values:
      This function always returns a 1 to tell parsing to stop.
---------------------------------------*/

int yywrap()
   {
   time_t endTime;

   printf("\nRead Statistics:\n\n");

   if (numberLexErrors + numberYaccErrors == 0) {
      endTime = time(NULL);
      printf("Elapsed parse\n");
      printf("   time (sec): %6d\n", endTime - startTime);
      printf("Nodes:       %8d\n", meshdata_.numberNodes);
      printf("Elements:    %8d\n", meshdata_.numberElements);
      }
   else
      printf("Read not completed because of error conditions.\n");

   /*  yywrap should return 1 to indicate successful completion
       and to tell yyparse to stop parsing. */

   return(1);

   } /* end yywrap */
```

B Yacc specification

```
%{

/*

File: yacc.spec
Description:
   Specify the grammar and the supporting functions for parsing a
   finite element input file.
External global variables:
   yylineno - The current line number of the parse.
   meshdata_ - Structure to hold the data read.
Global variables:
   startTime - Time parsing began.
   numberYaccErrors - Number of errors encountered during parsing.
Functions (functions appear in alphabetical order):
   InitGlobalVars - Initialize all global variables.
   InstallElement - Process element data.
   InstallNode - Process node data.
   readmesh_ - The main driver parser driver.  This is called from a Fortran
      module (the reason for the trailing _).
   yyerror - Performs actions when errors are encountered.

*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <time.h>
#include "parser.h"

/* External global variables: */

extern int yylineno;
extern MeshStruct meshdata_;

/* Global variables: */

int numberYaccErrors;
time_t startTime;

/* Function prototypes: */

static void InitGlobalVars();
static void InstallElement();
static void InstallNode();
void readmesh_();
void yyerror();

%}

%union {
   double real;
   int integer;
   char string[MAX_LINE_LENGTH];
   }

%token NODES
%token ELEMENTS
%token <integer> INTEGER
%token <real> REAL
%token <string> STRING
```

```
%start startpoint

%%

startpoint : items
           | startpoint items
           ;

items : NODES node_list
      | ELEMENTS element_list
      ;

element_list : INTEGER INTEGER REAL INTEGER INTEGER INTEGER {
                   InstallElement($1, $4, $5, $6);
                   }
             | element_list INTEGER INTEGER REAL INTEGER INTEGER INTEGER {
                   InstallElement($2, $5, $6, $7);
                   }
             ;

node_list : INTEGER REAL REAL REAL {
                InstallNode($1, $2, $3, $4);
                }
          | node_list INTEGER REAL REAL REAL {
                InstallNode($2, $3, $4, $5);
                }
          ;

%%

/*==========================================================================
   InitGlobalVars
   Purpose:
      Initialize all global variables.
   Global variables:
      numberYaccErrors - Counts number of parsing errors discovered.
      startTime - Record start time of parse.
      meshdata_.numberNodes - The number of nodes encountered.
      meshdata_.numberElements - The number of elements encountered.
--------------------------------------*/

static void InitGlobalVars()
   {

   numberYaccErrors = 0;
   startTime = time(NULL);
   meshdata_.numberNodes = 0;
   meshdata_.numberElements = 0;

   } /* end InitGlobalVars */


/*==========================================================================
   InstallElement
   Purpose:
      Add an element to the storage structure.
   Global variables:
      meshdata_ - Structure to hold data read.
   Local variables:
      elementID - The element number.
      node1, node2, node3 - Nodes comprising the element.
--------------------------------------*/

static void InstallElement(elementID, node1, node2, node3)
```

```
   int elementID;
   int node1, node2, node3;
   {

   meshdata_.node1[elementID-1] = node1;
   meshdata_.node2[elementID-1] = node2;
   meshdata_.node3[elementID-1] = node3;
   ++meshdata_.numberElements;

   } /* end InstallElement */


/*============================================================================
   InstallNode
   Purpose:
      Add a node to the storage structure.
   Global variables:
      meshdata_ - Structure to hold data read.
   Local variables:
      nodeID - The node identification number.
      x, y, z - The nodes x, y, and z coordinates.
--------------------------------------*/

static void InstallNode(nodeID, x, y, z)
   int nodeID;
   float x, y, z;
   {

   meshdata_.x[nodeID-1] = x;
   meshdata_.y[nodeID-1] = y;
   meshdata_.z[nodeID-1] = z;
   ++meshdata_.numberNodes;

   } /* end InstallNode */


/*============================================================================
   readmesh_
   Purpose:
      Server as the main entry point for parsing.  This function is called
      by the Fortran driver routine.
--------------------------------------*/

void readmesh_()
   {

   InitGlobalVars();

   yyparse();

   }


/*============================================================================
   yyerror
   Purpose:
      Report parser errors.  This routine is called by yyparse.
   Local variables:
      s - Error message passed in by the parser.
--------------------------------------*/

void yyerror(s)
   char *s;
   {
```

```
fprintf(stderr, "%s\n", s);

} /* end yyerror */
```

C Fortran parser driver

```
      program parser

      parameter (nodes=80000)
      parameter (elements=80000)
      integer numberNodes, numberElements
      real x(nodes), y(nodes), z(nodes)
      integer node1(elements), node2(elements), node3(elements)
      common /meshdata/ numberNodes, numberElements, x, y, z, node1,
     &   node2, node3

      call readmesh()

      end
```

D Miscellaneous code definitions

```
/*

File: parser.h
Description:
   This file defines the length of a line and also the
   structure the data will be stored in.  This structure
   must match the one defined in the common block in the
   Fortran driver code.

*/


#define MAX_LINE_LENGTH 80
#define NUM_NODES 80000
#define NUM_ELEMENTS 80000

typedef struct {
   int numberNodes;
   int numberElements;
   float x[NUM_NODES];
   float y[NUM_NODES];
   float z[NUM_NODES];
   int node1[NUM_ELEMENTS];
   int node2[NUM_ELEMENTS];
   int node3[NUM_ELEMENTS];
   } MeshStruct;
```

NO. OF COPIES	ORGANIZATION
2	DEFENSE TECHNICAL INFO CTR ATTN DTIC DDA 8725 JOHN J KINGMAN RD STE 0944 FT BELVOIR VA 22060-6218
1	DIRECTOR US ARMY RESEARCH LAB ATTN AMSRL OP SD TA 2800 POWDER MILL RD ADELPHI MD 20783-1145
3	DIRECTOR US ARMY RESEARCH LAB ATTN AMSRL OP SD TL 2800 POWDER MILL RD ADELPHI MD 20783-1145
1	DIRECTOR US ARMY RESEARCH LAB ATTN AMSRL OP SD TP 2800 POWDER MILL RD ADELPHI MD 20783-1145
	ABERDEEN PROVING GROUND
5	DIR USARL ATTN AMSRL OP AP L (305)

NO. OF COPIES	ORGANIZATION
2	UNIVERSITY OF MINNESOTA 125 MECHANICAL ENGRG ATTN PROF KUMAR TAMMA 111 CHURCH STREET SE MINNEAPOLIS MN 55455
2	UNIVERSITY OF DELAWARE MECHANICAL ENGRG DEPT 126 SPENCER LAB ATTN PROF SURESH ADVANI NEWARK DE 19716

NO. OF COPIES	ORGANIZATION
36	ABERDEEN PROVING GROUND DIR USARL ATTN AMSRL SC W. MERMAGEN R. LODER N. BOYER AMSRL SC C C. NIETUBICZ AMSRL SC CC P. DYKSTRA J. GROSH T. KENDALL C. ZOLTANI AMSRL SC CN D. TOWSON AMSRL SC S A. MARK M. BIEGA B. BODT M. TAYLOR AMSRL SC SM R. MOHAN D. SHIRES AMSRL SC SS V. TO C. HANSEN E. HEILMAN T. PURNELL K. SMITH M. THOMAS AMSRL SC SA J. WALL AMSRL SC A R. ROSEN AMSRL SC I E. BAUR J. GANTT M. HIRSCHBERG AMSRL SC II J. DUMER R. HELFMAN AMSRL IS TP B. BROOME S. CHAMBERLAIN B. COOPER A. DOWNS D. GWYN G. HARTWIG M. MARKOWSKI

USER EVALUATION SHEET/CHANGE OF ADDRESS

This Laboratory undertakes a continuing effort to improve the quality of the reports it publishes. Your comments/answers to the items/questions below will aid us in our efforts.

1. ARL Report Number ARL-TR-974 Date of Report March 1996

2. Date Report Received ______

3. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which the report will be used.) ______

4. Specifically, how is the report being used? (Information source, design data, procedure, source of ideas, etc.) ______

5. Has the information in this report led to any quantitative savings as far as man-hours or dollars saved, operating costs avoided, or efficiencies achieved, etc? If so, please elaborate. ______

6. General Comments. What do you think should be changed to improve future reports? (Indicate changes to organization, technical content, format, etc.) ______

CURRENT ADDRESS

Organization

Name

Street or P.O. Box No.

City, State, Zip Code

7. If indicating a Change of Address or Address Correction, please provide the Current or Correct address above and the Old or Incorrect address below.

OLD ADDRESS

Organization

Name

Street or P.O. Box No.

City, State, Zip Code

(Remove this sheet, fold as indicated, tape closed, and mail.)

(DO NOT STAPLE)